The FNC-2 system : advances in attribute grammars technology

Muriel Jourdan, Didier Parigot

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 834

THE FNC-2 SYSTEM : ADVANCES IN ATTRIBUTE GRAMMARS TECHNOLOGY

Martin JOURDAN
Didier PARIGOT

AVRIL 1988

RR 834

Table of contents, figures and tables

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

[MWW 84] Möncke, U., B. Weisgerber, and R. Wilhelm : "How to Implement a System for Manipulation of Attributed Trees", in *GI 8. Fachtagung "Programmiersprachen und Programentwicklung"*, Zürich, ed. U. Ammann, pp. 112-127, IFB vol. 77, Springer-Verlag (March 1984).

[Mye 84] Myers, E. W. : "Efficient Applicative Data Types", in *11th ACM POPL*, Salt Lake City, Ut, pp. 66-75 (January 1984).

[Par 85] Parigot, D. : "Un Système Interactif de Trace des Circularités dans une Grammaire Attribuée et Optimisation du Test de Circularité", rapport de DEA, Université de Paris-Sud, Orsay (September 1985).

[Par 87] Parigot, D. : "Mise en Œuvre des Grammaires Attribuées : Transformation, Evaluation Incrémentale, Optimisations", thèse de 3ème cycle, Université de Paris-Sud, Orsay (September 1987).

[Par 88] Parigot, D. : "Practical Transformation of Strongly Non-circular Attribute Grammars into l-ordered ones", manuscript submitted for publication, INRIA, Rocquencourt (March 1988).

[Rep 84] Reps, T. : *Generating Language-based Environments*, M.I.T. Press, Cambridge, Mass. (1984).

[Rii 83] Riis-Nielson, H. : "Computation Sequences : A Way to Characterize Subclasses of Attribute Grammars", *Acta Informatica* **19**, pp. 255-268 (1983).

[RSS 83] Räihä, K.-J., M. Saarinen, M. Sarjakovski, S. Sippu, E. Soisalon-Soininen, and M. Tienari : "Revised Report on the Compiler Writing System HLP78", report A-1983-1, DCS, University of Helsinki (January 1983).

[Sou 87] Souah, A. : "Système de Transformation d'Arbres Attribués : Etude des Principaux Systèmes et Spécification d'un Nouveau Système", rapport de DEA, Université d'Orléans (September 1987).

[VM 83] Vooglaid, A.O., and M. P. Méristé : "Abstract Attribute Grammars", *Progr. Comput. Soft.*, **8**, pp. 242-251 (1983).

[Yel 84] Yellin, D. : "A Survey of Tree-Walk Evaluation Strategies for Attribute Grammars", report, DCS, Columbia University, New York, NY (September 1984).

[Gie 86] Giegerich, R. : "On the Relation between Descriptional Composition and Evaluation of Attribute Coupled Grammars", Forschungsbericht nr. 221, Fachbereich Informatik, Universität Dortmund (July 1986).

[GJR 87] Garcia, J., M. Jourdan, and A. Rizk : "An Implementation of PARLOG Using High-Level Tools", in *ESPRIT 87 : Achievements and Impact*, Brussels, ed. Commission of the European Communities — DG XIII, pp. 1265-1275, North-Holland (September 1987).

[JOR 75] Jazayeri, M., W. F. Ogden and W. C. Rounds : "The Intrinsically Exponential Complexity of the Circularity Problem for Attribute Grammars", *CACM* **18**, 12, pp. 679-706 (December 1975).

[Jou 84a] Jourdan, M. : "Strongly Non-Circular Attribute Grammars and their Recursive Evaluation", in *ACM SIGPLAN Symp. on Compiler Construction*, Montréal, published as *SIGPLAN Notices* **19** (6), pp 81-93 (June 1984).

[Jou 84b] Jourdan, M. : "Les Grammaires Attribuées : Implantation, Applications, Optimisations", thèse DDI, Université Paris VII (May 1984).

[JP 88a] Jourdan, M. and D. Parigot : "The FNC-2 System User's Guide and Reference Manual", release 0.3a, manuscript, INRIA, Rocquencourt (February 1988).

[JP 88b] Jourdan, M. and D. Parigot : "More on Speeding up Circularity Tests for Attribute Grammars", rapport RR-828, INRIA, Rocquencourt (April 1988).

[Jul 86] Julié, C. : "Optimisation de l'Espace Mémoire pour les Compilateurs Générés selon la Méthode d'Evaluation OAG : Étude des Travaux de Kastens et Propositions d'Améliorations", rapport de DEA, Département d'Informatique, Université d'Orléans (September 1986).

[Kas 80] Kastens, U. : "Ordered Attribute Grammars", *Acta Informatica* **13**, 3, pp. 229-256 (1980).

[Kas 84] Kastens, U. : "The GAG-System — A Tool for Compiler Construction", in *Methods and Tools for Compiler Construction*, ed. B. Lorho, pp. 165-182, Cambridge University Press (1984).

[Kas 87] Kastens, U. : "Lifetime Analysis for Attributes", *Acta Informatica* **24**, 6, pp. 633-652 (November 1987).

[KHZ 82] Kastens, U., B. Hutt, and E. Zimmermann : *GAG A Practical Compiler Generator*, LNCS vol. 141, Springer-Verlag (1982).

[KLM 83] Kahn, G., B. Lang, B. Mélèse, and E. Morcos : "Metal : a Formalism to Specify Formalisms", *Science of Computer Programming* **3**, pp. 151-188 (1983).

[Knu 68] Knuth, D. E. : "Semantics of Context-free Languages", *Mathematical Systems Theory* **2**, 2, pp. 127-145 (June 1968). Correction : *Mathematical Systems Theory* **5**, 1, pp. 95-96 (March 1971).

[KW 76] Kennedy, K., and S.K. Warren : "Automatic Generation of Efficient Evaluators for Attribute Grammars", in *3rd ACM POPL*, Atlanta, Ge, pp. 32-49 (January 1976).

[LP 75] Lorho, B. and C. Pair : "Algorithms for Checking Consistency of Attribute Grammars", in *Proving and Improving Programs*, eds. G. Huet and G. Kahn, Colloque IRIA, Arc-et-Senans, pp 29-54 (1975).

[Mön 87] Möncke, U. : "Grammar Flow Analysis", ESPRIT PROSPECTRA Project report S.1.3.–R–2.2, Universität des Saarlandes, Saarbrücken (January 1987). Submitted for publication.

[Boc 76] Bochmann, G. V. : "Semantic Evaluation from Left to Right", *CACM* **19**, 2, pp. 55-62 (February 1976).

[CF 82] Courcelle, B. and P. Franchi-Zannettacci : "Attribute Grammars and Recursive Program Schemes (I and II)", *Theoretical Computer Science* **17**, 2 and 3, pp. 163-191 and 235-257 (1982).

[Che 81] Chebotar, K.S. : "Some Modifications of Knuth's Algorithm for Verifying Cyclicity of Attribute Grammars", *Progr Comput. Soft* **7**, pp 58-61 (1981).

[Che 85] Chebotar, K.S. : "Cyclicity in Attribute Grammars : Practical Approach", French-Soviet Colloquium on Computer Science, Tallinn, USSR (1985).

[Der 84] Deransart, P. : "Validation des Grammaires d'Attributs", thèse d'Etat, Université de Bordeaux I (October 1984).

[DJL 84] Deransart, P., M. Jourdan, and B. Lorho : "Speeding up Circularity Tests for Attribute Grammars", *Acta Informatica* **21**, pp 375-391 (1984).

[DJL 88] Deransart, P., M. Jourdan, and B. Lorho : *Attribute Grammars : Main Results, Existing Systems, Classified Bibliography*, to be published by Springer-Verlag in LNCS series (May 1988).

[DMR 87] Despland, A., M. Mazaud, and R. Rakotozafy : "Code Generator Generation based on Template-driven Target Term Rewriting", in *Rewriting Techniques and Applications*, Bordeaux, ed. P. Lescanne, pp. 105-120, LNCS vol. 256, Springer-Verlag (1987).

[EF 82] Engelfriet, J., and G. Filé : "Simple Multi-Visit Attribute Grammars", *JCSS* **24**, pp 283-314 (1982).

[Eng 84] Engelfriet, J. : "Attribute Grammars : Attribute Evaluation Methods", in *Methods and Tools for Compiler Construction*, ed. B. Lorho, pp 103-135, Cambrige University Press (1984).

[Far 82] Farrow, R. : "LINGUIST-86 Yet Another Translator Writing System based on Attribute Grammars", in *ACM SIGPLAN'82 Symp. on Compiler Construction*, Boston, Mass., published as *SIGPLAN Notices* **17**, 6, pp. 160-171 (June 1982).

[Far 84] Farrow, R. : "Sub-Protocol Evaluators for Attribute Grammars", in *ACM SIGPLAN'84 Symposium on Compiler Construction*, Montréal, published as *SIGPLAN Notices* **19**, 6, pp. 70-80 (June 1984).

[Fil 83] Filé, G. : "Interpretation and Reduction of Attribute Grammars", *Acta Informatica* **19**, pp 115-150 (1983).

[Fil 87] Filé, G. : "Classical and Incremental Attribute Evaluation by Means of Recursive Procedures", *Theoretical Computer Science* **53**, pp. 25-65 (January 1987).

[Gan 82] Ganzinger, H. : "An Overview of the Attribute Definition Language ADELE", in *GI 3. Fachgespräch "Compiler-Compiler"*, München, ed. W. Henhapl, pp. 22-53 (March 1982).

[GB 85] Gombás, E. and M. Bartha : "A Multi-visit Characterization of Absolutely Non-circular Attribute Grammars", *Acta Cybernetica* **7**, pp. 19-31 (1985).

[GG 84] Ganzinger, H. and R. Giegerich : "Attribute Coupled Grammars", in *ACM SIGPLAN'84 Symp. on Compiler Construction*, Montréal, published as *SIGPLAN Notices* **19**, 6, pp. 157-170 (June 1984).

[GGM 82] Ganzinger, H., R. Giegerich, U. Möncke, and R. Wilhelm : "A Truly Generative Semantics-directed Compiler Generator", in *ACM SIGPLAN'82 Symp. on Compiler Construction*, Boston, Mass., published as *SIGPLAN Notices* **17**, 6, pp. 172-184 (June 1982).

cess these modules were specified in OLGA and, although their first implementation is written by hand, using the OLGA text as a guideline, the subsequent versions will be constructed by FNC-2 itself. The first of these modules is the translator to C [Ayr 87].

We also intend in the near future to rewrite in OLGA the Pascal to P_code compiler [Jou 84b] and the Pascal to Ada translator [BDJ 84] which had previously been constructed by the FNC/ERN system, using Lisp as the base language. The latter AG is especially interesting since it was written with a very bad programming style (involving side-effects during attributes evaluation) to gain efficiency, and translating it into OLGA, which enforces the applicative style, will undoubtedly be a good experience.

Regarding performance, we unfortunately cannot give yet any figure. We can only say — and hope to be believed on word, since we are very confident — that the time and space efficiency of the evaluators generated by FNC-2 will be at least equal to that of the best AG systems presently available (no! we will cite no name...), and probably better, so that they will be able to compete with their hand-written equivalents.

9. CONCLUSION.

We have presented in this paper the new FNC-2 AG processing system, which gathers a number of advances in AG technology. Although none of them taken separately is a major breakthrough, we feel that their combination will result in a very effective tool which largely demonstrates the usefulness of the AG concept and should convince industrial companies to, at last, include AGs in their collection of programming aids.

Much work still needs to be done to reach production quality, but we believe that we are on the right way.

Acknowledgements : thanks go to Bernard Lorho for useful comments on a previous draft of this paper, and to Antoine Rizk for correcting our use of the English language. We would like to add special thanks to the "Attrisem" group of the "P.R.C. Programmation Avancée et Outils pour l'Intelligence Artificielle" for making possible useful contacts with other people interested in attribute grammars.

10. REFERENCES.

[Ayr 87] Ayrault, C. : "Implantation en C des Structures de Données du Langage OLGA", rapport de DEA, Université d'Orléans (September 1987).

[Bar 84] Barbar, K. : "Classification des Grammaires d'Attributs Ordonnées", rapport 8412, Université de Bordeaux I (April 1984).

[BCD 87] Borras, P., D. Clément, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang and V. Pascual : "CENTAUR : the System", rapport RR 777, INRIA, Sophia-Antipolis (December 1987).

[BDJ 84] Boullier, P., P. Deschamp, and M. Jourdan : "Application d'Outils de Haut Niveau à la Réalisation d'un Traducteur Automatique Pascal-Ada", in *2ème Coll. AFCET de Génie Logiciel*, Nice, pp. 375-385 (June 1984).

[BDL 88] Boullier, P., P. Deschamp and B. Lorho : *The SYNTAX Reference Manual*, to appear, INRIA, Rocquencourt (1988).

piler writing system embedding FNC-2, an interface with the scanner and parser generator SYNTAX has been realized.

To each production of the concrete grammar is associated a tree construction expression written in a language resembling a subset of OLGA. This is as if each non-terminal had a single, tree-valued, synthesized attribute, which can be evaluated in parallel with parsing since SYNTAX uses the LALR(1) bottom-up parsing method. The trees which are constructed must conform to a given attributed[15] abstract syntax, the operators of which act as tree construction functions. User functions may also be defined. The other constructs of this tree construction language include a conditional, the **let** and **message** operators, a number of pattern-matching constructs to restructure already constructed trees, and a **map tree** operator to process forests obtained by destructuring a list node.

The expressive power of this language has proven very important, allowing for instance the transformation of OLGA operators into function calls and, less trivially, the "bubbling" of embedded **case position** constructs up to the semantic rule level, involving the application of complicated distributivity rules. See [JP 88a] for more details.

7.4. Implementation

FNC-2 is (being) written in C and in OLGA and (will) run under UNIX[16] on SUN-3 machines and probably others ; the main machine dependencies lie in the graphic interface of the interactive circularity trace system, but the other parts of the system should prove easily portable.

8. APPLICATION AND PERFORMANCE.

This section will be probably very disappointing to the reader, because FNC-2 is still under development and has not yet generated anything, not even the famous binary numbers example of [Knu 68]. However, since the OLGA language is in a rather stable state, a number of AGs have already been written, and the corresponding evaluators will be generated as soon as the first implementation of FNC-2 is available, which is foreseen for September '88.

The OLGA Reference Manual in [JP 88a] contains a two-phase AG describing the static semantic check of the toy language *simproc* and its translation (using tree construction functions) into terms of an abstract data type defined by an attributed abstract syntax. It also contains a parameterized declaration module defining a symbol-table abstract data type for block-structured languages with two corresponding definition modules, one based on linear lists and the other on binary trees, using non-destructive operations [Mye 84]. Although this is a simple example, it fully demonstrates the qualities of OLGA.

A more realistic example is a three-phase compiler of the parallel logical language Parlog into code for the Sequential Parlog Machine [GJR 87], developed in the ESPRIT project "COCOS". This 3500+ lines work was completed by an untrained programmer, acquainted neither with Parlog nor with attribute grammars, in less than six months.

We intend to bootstrap the components of FNC-2 which allow it, namely the abstract syntax compiler, the OLGA reader and checker and the translators. Very early in the development pro-

---

[15] Only lexical attributes are allowed here.

[16] UNIX is a registered trademark of AT&T Bell Laboratories.

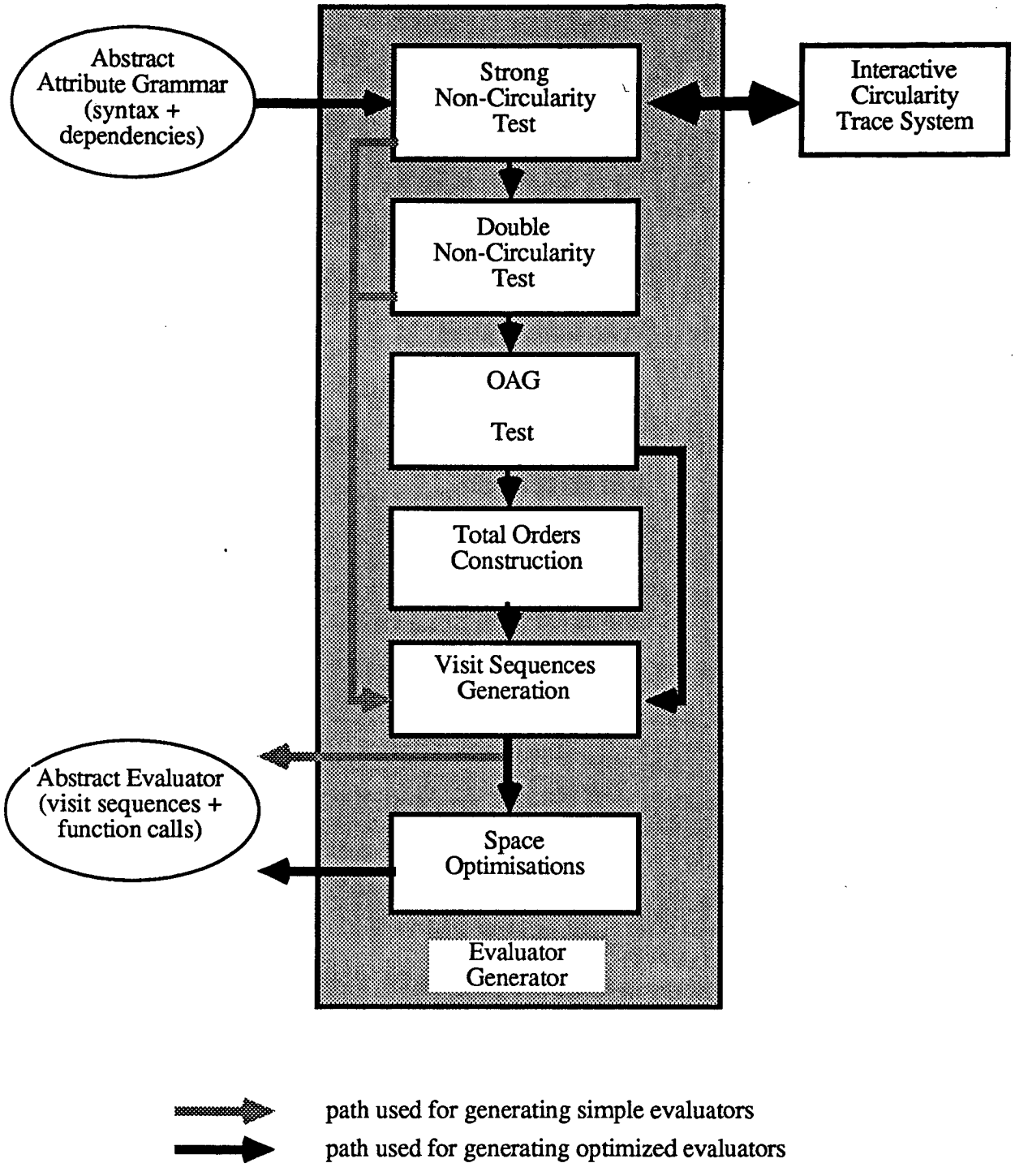


Figure 4 — the evaluator generator

7.3. Input tree construction

Since FNC-2 generates evaluators as tree-to-tree mappings, it is not concerned by the way those trees are constructed in the first place. However, as the first step towards a complete com-

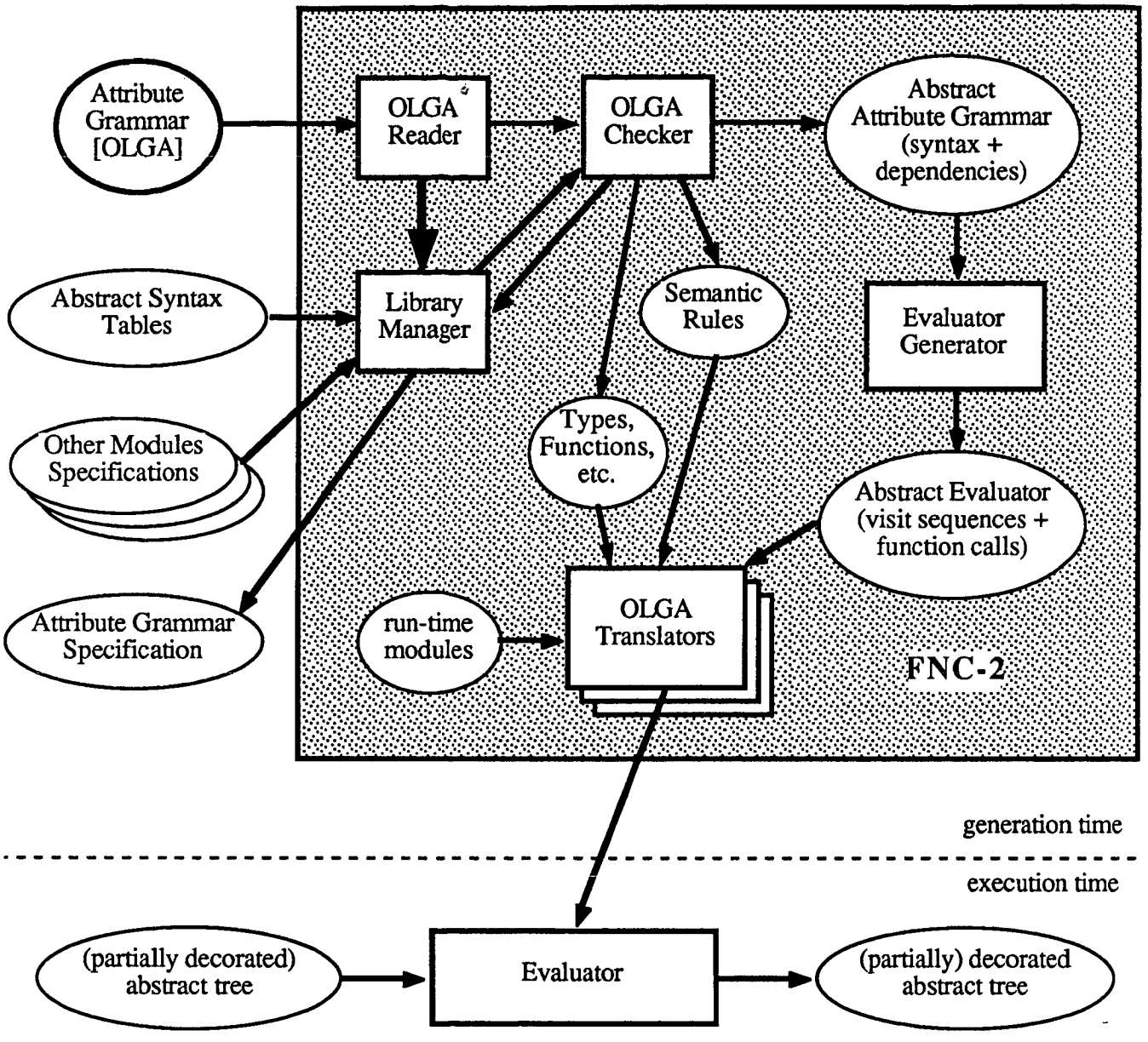


Figure 3 — the FNC-2 system

The abstract evaluator and the other components of the AG (types, functions, semantic rules) are then input to one of several translators which perform the final code generation into the chosen implementation language. Well-known compilation techniques are applied in this phase to produce efficient code. At the time of writing only the translator to the C language has been designed [Ayr 87]. Garbage collection — necessary to save space in an applicative language — uses, for the sake of simplicity, a reference count scheme. The translator to Lisp will come in the near future and will be much simpler since it will use the built-in garbage collector.

with modules generated by other tools which adopt the same philosophy : upstream, a scanner and parser such as the SYNTAX[14] system [BDL 88], or a programming environment generator such as Centaur [BCD 87] ; downstream an attributed tree transformation system such as OPTRAN [MWW 84] or Transat [Sou 87], or a code generator generator (e.g. [DMR 87]).

The independence of an AG specification in OLGA w.r.t. to its implementations also allows FNC-2 to remain a research tool with which new concepts can be experimented.

7.2. Operation

The structure of FNC-2 is depicted in figures 3 and 4. A separate processor, not shown in the figures, is in charge of compiling attributed abstract syntaxes descriptions into an internal tabular form. An OLGA program, be it a simple module or a complete AG, is input to the OLGA reader, which scans and parses it and searches it for references to external modules, including attributed abstract syntaxes. These references are resolved by a library manager which maintains the consistency of mutually referencing modules. The OLGA checker performs static semantics analysis and constructs a list of exported objects to be used by the library manager. If the input text is a declaration module, then the work is done, else if it is a definition module, then code is generated (see further), else — this must be an AG — more work is needed before the final code generation.

First the well-definedness of the AG is checked and missing attribute declarations and semantic rules are generated. Then an abstract AG scheme is constructed, which contains only the syntax — in a more "concrete" form, in particular regarding list operators, so that the algorithms of sections 2 to 5 be more readily applicable —, the attributes attachments and the dependencies extracted from the semantic rules (including those defining local attributes). This abstract AG scheme is input to the evaluator generator (see figure 4).

The evaluator generator behavior varies according to the desired kind of evaluator (evaluation mode and optimization level). In any case however, it starts with the SNC test which constructs one *pop* for each non-terminal. If a circularity is detected, the interactive circularity trace system is activated to search the "erroneous" semantic rules and the generation is thereafter aborted. Otherwise, and if a simple non-deterministic exhaustive evaluator is demanded, visit sequences are directly constructed ; if a simple incremental evaluator is chosen, this construction is preceded by the DNC test.

In the other cases the DNC and OAG tests are performed. If the latter fails the AG is transformed into an l-ordered one. In all cases the result is a (set of) totally-ordered partition(s) for each non-terminal. If the exhaustive evaluation mode with space optimizations has been chosen, the temporary attributes are determined, their lifetime computed and their storage mode (global variable, stack or tree node field) is determined. Then comes the construction of visit-sequences according to the chosen evaluation mode, and space management instructions are added if necessary. This completes the construction of the abstract evaluator.

FNC-2 can be directed to generate one evaluator for each AG, or even one for each pass of each AG, for debugging purposes, or to (try to) generate a single evaluator for a sequence of AGs to gain efficiency.

---

[14] SYNTAX is a registered trademark of INRIA.

6.9. Conclusion

The present definition of the OLGA language is still undoubtedly perfectible, and it is actually under continuous revision. We however feel that the design goals we have stated have been met, and preliminary experience with OLGA (see section 8 below) is very encouraging.

The OLGA Reference Manual is part II of [JP 88a].

7. THE FNC-2 SYSTEM

7.1. Functionalities

The FNC-2 attribute grammar processing system embodies all of the above described features and aims at expressive power, efficiency, ease of use and versatility.

The *expressive power* is due to the AG class it accepts, the very large SNC class, which encompasses all the practical AG examples, in particular those encountered in compilation. Furthermore, this theoretical expressive power is enforced in practice by two features :

- an interactive circularity trace system [Par 85] is associated with the SNC test and allows, when an AG fails to be SNC, to discover exactly the semantic rules which are involved in the circularity and correct only those which are "erroneous" without rewriting the whole AG ; this allows to take full advantage of the SNC class ;
- the possibility to "pipe" several AGs, either on the same syntax or in the manner of ACGs, allows to describe arbitrarily complicated dependencies while keeping each AG in the SNC class.

The *efficiency* stems from the evaluation methods used in FNC-2, described in sections 2 and 3 above, and from the space optimizations of section 4, which result in fast and memory-saving evaluators. Efficiency is also present in the constructor itself, thanks to the construction-time optimizations of section 5, and to the fact that many of its components are bootstrapped.

The *ease of use* results from the qualities of the OLGA language.

The *versatility* stems from the conjunction of two features :

- an AG specification in OLGA is completely independent from its implementation(s), which can use several evaluation modes (exhaustive or incremental), several implementation languages (C, Lisp and Ada[12] are the first targets), several optimization levels (to save construction time when an AG is being developed, it is possible to construct a non-completely deterministic evaluator directly after the SNC test for exhaustive evaluation and directly after the DNC test for incremental evaluation [Fil 87]) and several machine architectures (sequential of course, but an implementation on a shared-memory multiprocessor system is being investigated) ;
- considering an AG as defining a tree-to-tree mapping allows to combine the resulting evaluators in multiple ways as a Meccano[13] game (e.g. by adding a source-level optimization phase between the front-end and the intermediate code generator of a compiler), and also

[12] Ada is a registered trademark of the government of the U.S.A., AJPO.

[13] Meccano is a registered trademark.of Meccano S.A.

nodes. To define synthesized attributes of a list node, a **case arity** construct distinguishes when the node has zero, one or more sons and restricts accesses to the **first** and **last** son only. A **map** construct may also be used on list operators to compute some value by traversing the list of the sons and applying at each node some function using the already accumulated value.

The subtrees hanging from RHS nodes may also be examined using pattern-matching, and their attributes are thereafter accessible. This allows to acquire syntactic information without propagating special-purpose attributes in a cumbersome way.

In most cases attributes need not be declared at all, and copy rules need not be written since default rules allow to generate missing declarations and semantic rules automatically ; this is borrowed from the ancestor of FNC-2, the FNC/ERN system [Jou 84b], where it proved very useful.

6.8. Structure and modularity

OLGA is a block-structured language and provides a high level of modularity (req. 1). Constants, values (computed at run-time), types, functions and exceptions may be declared in separate declaration modules and implemented in definition modules. Externally-defined objects can be imported in any block. Modules may have types and/or functions as parameters. Opaque types are supported, so that abstract data types can be defined.

An AG is considered as a mapping from some attributed abstract tree to zero[10], one or several attributed abstract trees. The input and output domains of an AG are hence described by (separate) attributed abstract syntaxes, which play the role of declaration modules. This allows to "pipe" several AGs.

An AG itself may be structured into passes, phases and of course productions. Each pass is really an AG with its own attributes, but all the passes of a given AG work on the same input abstract tree. A phase is just a syntactic structuring device and has no semantics, except that it is a block and may thus contain local declarations. Each production is also a block with possibly local declarations ; in particular local values can depend on input attribute occurrences and thus play the role of "local attributes". The semantic rules of a given production in a given pass may be spread out over several phases, which improves the overall structure ; however the well-definedness is tested only at the pass or complete AG level.

Output attributed abstract syntaxes of a given AG are considered as tree types (for phyla) and tree construction functions (for operators). Phyla of the input syntax are given one or several (synthesized) tree-typed attributes and tree construction functions are used in semantic rules defining them, so that the result of the evaluation is the set of tree-typed attributes of the root[11] ; the input tree is destroyed ; this defines a "functional" AG. Alternatively, when there exists only one output attributed abstract syntax, the input and output ones may differ only in the set of attributes attached to the phyla ; the result of the evaluation is then the whole unmodified tree with the new attributes attached (input attributes which do not appear in the output syntax are discarded) ; this defines a "side-effect" AG.

---

10 When no output abstract syntax is declared for a given AG, the result of the evaluation is an anonymous record containing the synthesized attributes of the root.

11 This allows to express, if not to implement, *attribute coupled grammars* (ACGs) [GG 84, Gie 86]. The implementation of descriptional composition is planned in the future.

be (mutually) recursive. It is also possible to declare and use functions written in a foreign language[9].

6.5. Expressions

Expressions include manifest constants, record field access, set constructor, function calls, a number of unary and binary operators (which are considered as functions with a fixed number of arguments), boolean short-circuit operators, a **let** construct for naming subexpressions, a construct for emitting (error) messages, and exception raising and catching. Exceptions may have parameters.

Selection constructs include a conditional with **elsif**, a selection based on scalar value (**case** ... **is** ... **end case**), a selection based on pattern-matching (**case** ... **match** ... **end case**) and a selection based on the tag of a union-typed value (**case type** ... **is** ... **end case**).

6.6. Abstract syntax

The syntactic base of an AG in OLGA and in the FNC-2 system is an *abstract syntax* rather than a concrete one as in the original theory. This has three primary advantages :

- the AG writer gets rid of clumsy syntactic constraints due to the parsing method used, such as the priority of operators or the well-known "dangling else" problem ;
- abstract syntax trees are generally smaller than concrete ones because purely syntactic information such as keywords is omitted ;
- an abstract syntax allows to outline semantic notions more clearly than concrete syntax ; for instance in OLGA the operators are mere function calls and hence appear as such in the abstract syntax.

An abstract syntax is composed of *phyla*, corresponding to concrete non-terminals, and *operators*, corresponding to productions. Operators are either heterogeneous, fixed-arity ones or homogeneous, variable-arity ones (*list operators*, either possibly empty or never empty). This formalism is borrowed, with minor modifications, from Metal, the definition language of the programming environment generator Mentor [KLM 83]. With each phylum is associated an implicitly-defined nullary operator which can be used in various "erroneous" situations such as unrecoverable syntactic errors if the input tree is constructed by an off-line parser or at yet unexpanded nodes if the tree is constructed with the help of a syntax-directed editor.

6.7. Attributes and semantic rules

Attributes are attached to phyla, and attribute occurrences and semantic rules to operators (productions). Semantic rules appear as the assignment to some attribute occurrence of some expression, possibly involving other attribute occurrences, including attributes of nodes upward in the tree. An AG must be in normal form [Boc 76] (but see next subsection for "local attributes").

Special constructs, derived from those of [VM 83], deal with attributes of list operators. To define inherited attributes of the sons of a list node, a **case position** construct distinguishes the **first** and/or **last** and the **other** nodes, and restricts access to the immediately neighboring

---

[9] This is useful to express simply some notions of "states" or "global values" which would be cumbersome with pure AGs, for instance the generation of fresh labels to produce assembly code.

3) it should be independent from any particular implementation, but allow for efficient implementation on conventional machines ;

4) its learning should be easy ;

5) its semantics should be clear, unambiguous and well-defined ;

6) and, when all of the above requirements are satisfied, it should remain simple.

OLGA is thus a completely original language designed uniquely for AG specification, and not an existing language augmented with notations for dealing with attributes (requirements 2 and 3). Very few AG languages were so designed (see part II of [DJL 88]). The most achieved trials in this respect are probably ADELE [Gan 82], the language of MUG2, and ALADIN [KHZ 82], the one of GAG. Actually both of them had important influence on the design of OLGA.

6.2. Applicativeness

OLGA is a *fully applicative* language, to comply with requirements 1 and 2, which means that there exist no variables — just values and functions — and that side-effects are impossible (apart from the emission of (error) messages). However OLGA is not a functional language with higher-order functions, since, in our opinion, this is not (yet) amenable for efficient compilation and execution (req. 3).

6.3. Types

OLGA is *strongly typed* (req. 1), however this does not impair the ease of writing since, due to a powerful *type inference* mechanism for type-checking functions and expressions, type declarations for attributes and function parameters are usually not needed. A restricted form of *polymorphism* and full function names *overloading* are also provided.

OLGA provides the following basic types : integer, real, boolean, character, (character) string, token (lexical units of the input text or tree), source-index (position in the source text, to attach messages), and *void* as in Algol 68. Type constructors include enumeration, sets (of scalar values), records, (tagged) unions and (homogeneous) lists. Types may be (mutually) recursive[7], but OLGA does not support infinite values. Each type must be named. With each basic or declared type are associated one or more conversion (for scalar types) or construction (for declared types) functions with the same name as the type[8]. Subranges of scalar types may also be declared, but they are merely considered as constraints to be dynamically checked, e.g. when values are passed as parameters or assigned to attributes occurrences.

6.4. Functions

Functions are declared as usual, except that their body is an expression rather than a statement list. Parameters are passed by value (what is a "reference" in an applicative language?). Type inference usually eliminates the need to declare parameter types. Functions may of course

---

[7] the type *void* being generally used to terminate the recursion.

[8] Overloading is heavily used therefore.

DNC = double non-circularity test with our own improvements, after the SNC test has been performed.

The tcl_{circ} figure is interesting since, being the number of final graph combinations, it is a very good measure of the "semantic complexity" of a given AG. We added statistics for strong and double non-circularity tests, both as a comparison point and because all our examples are doubly non-circular. The rough results are presented in table 3 below.

The first comment which comes to mind when reading this table is that our improvements are *always* effective ; the gain on the number of transitive closures is even very impressive for large examples (about 3.5 times for Pascal).

It is impossible to prove formally that our algorithms are optimal, and we won't claim that either. Let us however examine the processing of a "simple" subgrammar : we need (at least) one iteration for constructing the relations, one iteration to check that convergence is reached and, one iteration for actually testing non-circularity ; thus, for a "simple" grammar, the total number of transitive closures is of the order of three times the number of transitive closures needed for testing non-circularity. With our improvements we reach this behavior for three of our examples, including one large grammar (PL/1-C) ; this is, in our opinion, an indication that A7 is near-optimal.

		simula	PL/1-C	pascal
A4	*tcl*	2846	1871	7735
	cpu	98.1	121.	523.
A7	*tcl*	1511	995	2228
	cpu	70.7	96.1	182.
	tcl_{circ}	272	389	457
SNC	*cpu*	15.	17.	23.
DNC	*cpu*	9.	17.	13.

Table 3 — practical comparison of non-circularity test algorithms.

For more details on our improvements see [Par 85, JP 88b].

6. THE OLGA LANGUAGE

6.1. Rationale

The main design goals of OLGA[6], the AG definition language of FNC-2, are as follows :

1) it should embody all the concepts necessary to provide the AG writer with complete programming safety and high productivity ;

2) it should offer all the expressive power of AGs, but no more, and remain close to the original theory so that the above described evaluation methods are readily applicable ;

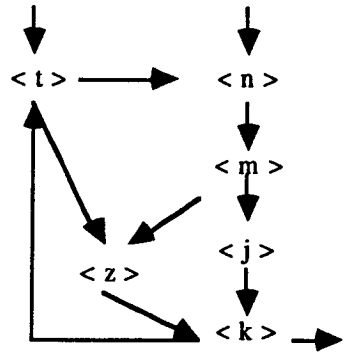
[6] OLGA is the acronym of the French wording « Ouf! un Langage pour les Grammaires Attribuées », which roughly means « At last! a Language for AGs ».

relation Γ (and Γ^{-1}), so it is *a priori* impossible to find a total order compatible with Γ^{-1} ; heuristics will help make the choice.

We can define in a natural way entry points and output points of a subgrammar [DJL 84]. The first heuristic is then to order the non-terminals from the output points to the entry points in each class ; it is natural to do so because the algorithm works bottom-up and the former non-terminals will be "lower" in derivation trees than the latter.

However this gives no indication about the relative ordering of output non-terminals of a given class. To solve this difficulty, it is also natural to take the relation Γ^{-1} into account, as shown by the following example :

Example : take the following class and relation Γ^{-1} :



< t > and < n > are output points, < k > is the entry point. It is obvious that < t > should precede < n > in the final ordering, so that information computed on < t > can be used on < n > in the same iteration. The final ordering is thus : < t >, < n >, < m >, < j > and < z >, < k >.

Our second heuristic is hence to order the entry points which do not directly derive in other entry points before the other.

This improvement may be used independently from semantic stability. It may also be used independently from grammar partitioning, but the results are then less effective because the entry and output points are not well defined ; also, non-terminals static ordering gives you grammar partitioning "for free", and it would be unfortunate to ignore the advantages of the latter, especially regarding space efficiency [Che 81].

5.3. Experimental results

We have implemented these improvements in the FNC/ERN system on Multics at INRIA, and tried them on the same real examples as for our transformation (see section 2). This section presents some statistics we gathered from their execution.

We will use the following notations :

A4 = non-circularity test with all the improvements presented in [DJL 84].
A7 = A4 algorithm with our own improvements.
tcl = total number of transitive closures (basic steps) computed (construction of the dependency relations plus actual non-circularity test, performed in a separate pass).
cpu = CPU time in seconds.
tcl_{circ} = number of transitive closures computed for testing only non-circularity.
SNC = strong non-circularity test with our own improvements.

We present these improvements, called *semantic stability* and *non-terminal static ordering*, only on the non-circularity test because this is the situation where they are most effective. Moreover, on the other algorithms, these improvements are only particular cases of the more general presentation on the non-circularity test. Lastly many authors have worked out other improvements to reduce the time and space complexity of the circularity test because of its worst case exponentiality [LP 75, Che 81, Che 85, DJL 84], so we may add our work to this long list. When we integrate all these results in a single implementation, our practical results show that we nearly reach the optimal behavior of this test on real AG examples. For the sake of brevity, we assume that the reader is familiar with the non-circularity test algorithm, particularly with the notion of dependency graph.

5.1. Semantic stability

Let us turn back to the standard algorithm in [Knu 68]. Its basic step chooses a combination of graphs in the set of relations attached to each non-terminal in the right-hand side of a production, builds the composite dependency graph, computes its transitive closure, tests its non-circularity, computes its projection onto the left-hand side non-terminal and, if necessary, adds the latter to the set of relations attached to that LHS non-terminal. This basic step must be repeated for each such combination, until convergence is reached, i.e. no new relation can be added.

The improvement discussed here, which we call *semantic stability*, is based on the observation that, if a given combination of graphs is "old" enough to have already been processed during a previous iteration, then it is useless to process it again — i.e. we may skip the above basic step —, because the information it conveys has already been added to the set of relations attached to the LHS non-terminal. Recall that the nature of this information (a set of binary relations on its attributes, expressing their mutual dependencies) is such that a single piece of information (a single dependency), once inserted, cannot be subsequently deleted.

Note that semantic stability may be used independently from, or in combination with, all the other improvements described in [DJL 84] or hereafter. Also note that semantic stability was devised by Chebotar [Che 85] independently from us.

5.2. Non-terminals static ordering

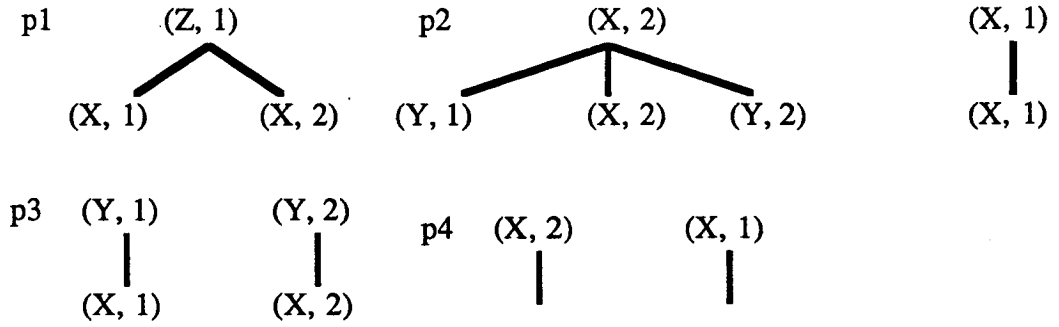
In any circularity test, the set of dependency graphs $D(X)$ attached to each non-terminal $X \in N$ depends on the dependency graphs $D(X_i)$ for each X_i in the right-hand sides of productions of which X is the left-hand side, i.e. for each X_i such that $X \ \Gamma \ X_i$[5]. In order that such information propagates faster, i.e. in order to reach convergence with less iterations, it is natural to take into account the syntactic dependency relation Γ to (statically) find an optimal order for processing the non-terminals.

The presentation of our ordering is more natural, and its effect is also greater, if it is associated to the grammar partitioning devised by Chebotar [Che 81, DJL 84]. In this case, we apply the ordering algorithm independently on each subgrammar (partition).

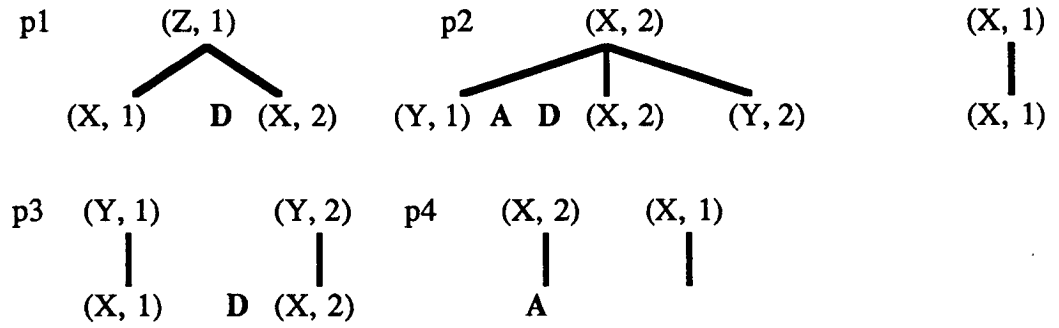
The ordering is achieved by topologically sorting the non-terminals according to the inverse relation Γ^{-1}. However, by definition, each subgrammar is a strongly connected component of the

[5] The relation Γ is the syntactic "derives in" relation between non-terminals [Che 81, DJL 84].

The G_V grammar :



The G(X.a) grammar :



From the example it is clear that the attribute X.a can be implemented as a global variable. But if we forget in the G_V-grammar the non-terminal indexing (X, k) due to the sequence visit number, then, in our example, the attribute X.a can't be implemented as a global variable. The indexing of the non-terminals of G_V is the major difference between our approach and that of Kastens, and the above example shows that our approach is also finer. We unfortunately don't yet have practical results of this approach.

Lastly, a word about another improvement over Kastens' work being pursued (but not yet completed) by Julié. After determining the implementation mode (global variable, stack or node field) of all the attributes, Kastens tries to group together two or more global variables or stacks [KHZ 82]. Unfortunately the implementation of this algorithm in the GAG system is very sensitive to the way the AG is written (order of attribute declarations and productions). C. Julié is investigating a more global and effective approach to this problem, using cost criteria rather than Kastens' simple feasibility criteria.

5. CONSTRUCTION-TIME OPTIMIZATIONS.

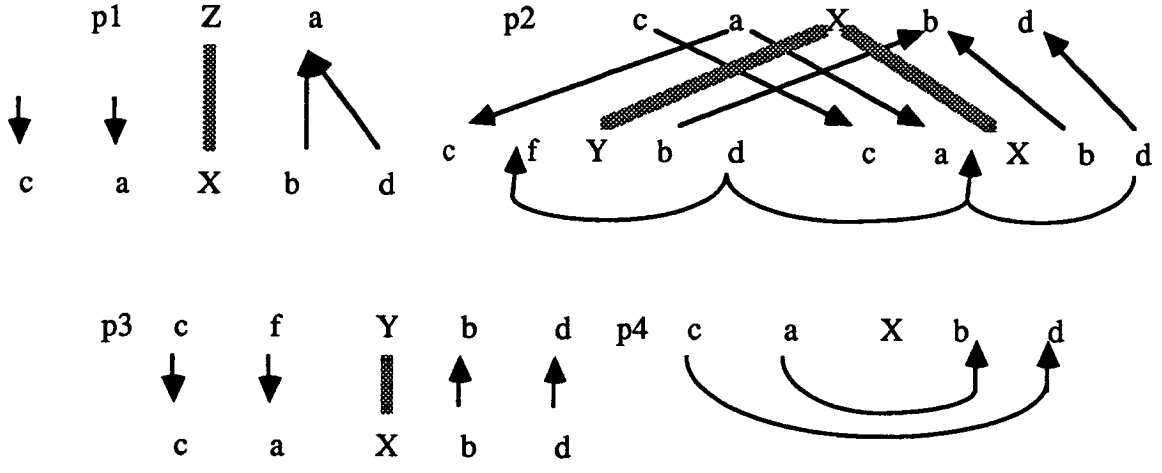
We now present two improvements which can be applied to the non-circularity test as well as to the SNC test, DNC test, and also to our SNC to l-ordered transformation[4]. Let us recall that the CPU time of these algorithms is very high, particularly for the non-circularity test, and that this may preclude their practical use. But considering the results we obtain when we include those optimizations, we can say that the construction time of our evaluators is very reasonable.

[4] These improvements are actually applicable to any Grammar Flow Analysis problem [Mön 87].

Then, for each attribute *a* of each non-terminal *X*, we use this G_V grammar to construct the G(X.a) grammar, by adding the two terminals, D for the definition of X.a, and A for the last use of X.a, as follows : for each visit sequence where X.a is used or defined, we add at the right place in the corresponding productions of G_V the terminal D when this attribute is defined and the terminal A when its last use occurs (see following example and [Kas 84, Jul 86]).

The conditions under which an attribute can be implemented as a stack or as a global variable can then be tested using these grammars G(X.a) : if the language L(G(X.a)) generated by one of these grammars is [DA]* (regular language), the attribute can be implemented as a global variable, and if L(G(X.a)) is well parenthesized, the attribute can be implemented as a stack.

Example : the AG (4 productions) :



The visit sequences :

```
sequence{a}-Z-p1 (tree) :
        eval {c} 1, p1 ;
        visit-S {d}-X- (son (1, tree)) ;
        eval {a} 1, p1 ;
        visit-S {b}-X-(son (1, tree))
        eval {a} 0, p1 ;
end sequence;

sequence{d}-X-p2 (tree) :
        eval {c} 2, p2 ;
        visit-S {d} -X- (son (2, tree));
        eval {d} 0, p2 ;
end sequence;

sequence{d}-Y-p3 (tree) :
        eval {c} 1, p3 ;
        visit-S {d}-X- (son (1, tree));
        eval {d} 0, p3 ;
end sequence;

sequence{b}-X-p2 (tree) :
        eval {c} 1, p2 ;
        visit-S {d}-Y- (son (1, tree));
        eval {a} 2, p2 ;
        visit-S {b}-X- (son (2, tree));
        eval {f } 1, p2 ;
        visit-S {b}-Y- (son (1, tree));
        eval {b} 0, p2 ;
end sequence;

sequence{b}-Y-p3 (tree) :
        eval {a} 1, p3 ;
        visit-S {b}-X- (son (1, tree));
        eval {b} 0, p3 ;
end sequence;

sequence{d}-X-p4 (tree) :
        eval {d} 0, p4 ;
end sequence;

sequence{b}-X-p4 (tree) :
        eval {b} 0, p4 ;
end sequence;
```

performed and the defined attribute instance is marked consistent, otherwise it is reevaluated and marked consistent according to the comparison of the old and new values.

The advantages of this method are the following : its conception is very simple, fully static, i.e without overhead, and purely recursive. Our first experimental results show that the incremental evaluators are very efficient. For more details see [Par 87].

4. SPACE OPTIMIZATIONS

Let us recall that the space consumption of the AG approach is certainly the most critical point hindering its industrial use. This explains that it has been the subject of many works (27 references in the bibliography of [DJL 88]), especially by Kastens [KHZ 82, Kas 84, Kas 87]. In this section we quickly present two extensions of Kastens' storage optimizations which are devised in [Jul 86, Par 87].

Kastens' approach is based on static attributes *lifetime analysis* to determine whether they can be implemented either in global variables or in stacks or finally in the tree. Note that to perform this analysis, we must know a total evaluation order, and this is the case for l-ordered AGs, in particular those resulting from our transformation. The results of Kastens' approach are very promising : on one of their examples, a Pascal compiler, the space of the attributes values is only a quarter of the space of the syntax tree [KHZ 82]. However these results can be improved.

The basic idea of this approach is that, for each attribute *a* of a non-terminal *X*, noted X.a, if the lifetimes of all its instances are disjoint, the attribute can be implemented as a global variable. If all these lifetimes are properly included or disjoint, the attribute can be implemented as a stack. In any other case the attribute must be attached to the tree nodes. To conduct lifetime analysis Kastens has defined a set of particular grammars, noted G(X.a), which allow to statically test the previous conditions.

The first extension [Jul 86] to Kastens' work is the following : if all the lifetimes of X.a are restricted to the visit of the node in which this attribute is created (this attribute is called *temporary*), this attribute can be implemented in a stack, without any other conditions. This is achieved by relaxing the (somewhat artificial) conditions imposed by Kastens on the management of those stacks. This result is important because, on our AG examples (*PL/1-C*, *Pascal*) as in most other examples, 90% of the attributes are temporary, and thus can be implemented at least as a stack or maybe as a global variable.

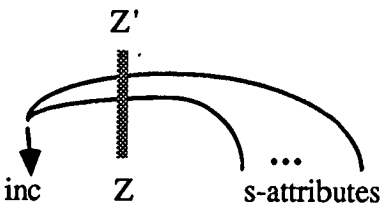
For the second extension [Par 87], we shall describe in a quick and informal way, by the following example, how we construct the particular grammars G(X.a). That construction will be a little different from Kastens' one. First we define a *visit grammar*, noted G_V, which is composed only with non-terminals of the form (X, k), where *X* is a non-terminal of AG, and *k* is an integer representing the visit number. This grammar G_V is constructed as follows :

- for each non-terminal X of the AG we define a set of non-terminals (X, k) of G_V such that each (X, k) represents the k^{th} visit of X ;
- for each visit sequence and each production of the original grammar, we construct a production of G_V such that the left-hand-side is (X_0, k_0) where k_0 is the number of this visit to X_0, and the right-hand side is a sequence of (X_i, k_i) representing the visits to the non-terminals in the RHS of the original production, in the order of the corresponding visit sequence.

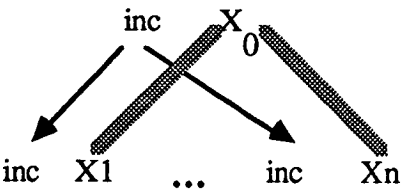
SNC closure condition for top-down evaluation and a new closure condition for bottom-up evaluation ; this defines the class of *doubly non-circular* AGs ; for more details see [Bar 84, Fil 87].

Using this set of recursive procedures, we can now compute all the attributes of node *u* (and some other attributes in *t'*) by calling the corresponding procedures in the order implied by $R_{DNC}(X)$: first, the i-attributes of *u* which depend on no other attribute of *u*, then the s-attributes of *u* which depend only on those i-attributes, and so on. However when this terminates, some attributes in *t'* have not been evaluated, namely the synthesized attributes of the nodes on the path from the root to *u* which depend on the last evaluated s-attributes of *u*. This is because, by definition, there are no i-attributes on the root (axiom of the AG). Starting at node *u*, the set of i-attributes procedures calls, allowing to climb up the tree, will not be able to reach the root. To solve this problem we transform the initial grammar by adding a particular i-attribute, noted *inc*, which will act as the root s-attributes on all the nodes (on each non-terminal of the grammar). Informally this transformation is as follows :

i) we add a particular production just at the beginning of the AG, above the axiom Z, with the following semantic rules :



ii) to each production of the AG we add the following semantic rules :



In all the graphs $R_{DNC}(X)$ the attribute *inc* will depend on all the other attributes and, hence, will be the last to evaluate using the process described above ; its evaluation at node *u* will then imply the evaluation of all attributes which were forgotten by the former process (except possibly attributes which do not contribute to the value of the attributes of the root ; if such attributes exist, this can be considered as an error in the design of the AG).

With this transformation and the corresponding family of graphs R_{DNC}, we can construct an evaluator composed of a set of recursive procedures for both i-attributes and s-attributes which will be able to start the evaluation at any node of the tree.

In order to make this method incremental, we have to add some "semantic control" to the evaluation process, allowing to stop the recursive evaluation whenever an old attribute instance proves to be consistent w.r.t. the new tree. This semantic control is implemented by two groups of consistency tests, which use a consistency marker for each attribute instance. The first group tests, before each call of a recursive procedure (for the i-attributes an the s-attributes), whether all of its attribute parameters[3] are consistent ; if so, and according to the relative position of the visited node and the modification point *u*, the call is not performed. The second group tests, before each call of a semantic rule, whether all its arguments are consistent ; if so, the call is not

[3] An evaluation procedure takes as parameters a tree node and some attribute values [CF 82, Jou 84a] ; the choice of the latter is based on the graphs R_{DNC} (*argument selector*).

pared to A1, a 10 % reduction of visits number, but the construction time is increased by about 50 %

• The run-time penalty induced by having a larger number of visit is very low : on a practical implementation, using our *Pascal* AG, we loose only 5% time when we increase the number of visits by 30% (this is not shown in our tables) ; this justifies the use of our inclusion orderings which, theoretically, tend to increase that number.

Thus, using one or the other topological sort algorithm is largely a matter of taste.

As a conclusion, let us recall that the good ideas of Kastens on memory optimization can only be applied if the AG is l-ordered ; unfortunately the membership test is NP-complete. Kastens, with his OAG approach [Kas 80], finds a proper subclass of l-ordered AGs in polynomial time. On the contrary, our transformation partially resolves that problem in two different ways. First, on some of the tested practical examples, the transformation shows in a reasonable construction time that the grammars are l-ordered (size ratio = 1). Secondly, when the transformation cannot show that the grammar is l-ordered, the size ratio remains close to one, still with a reasonable construction time.

3. INCREMENTAL EVALUATION OF SNC AGS

In this section we present an incremental evaluation method which is based on the method devised by Filé [Fil 87] for doubly non-circular (DNC) AGs. However, using the previous transformation — l-ordered AGs are a proper subclass of DNC AGs — our incremental method can be applied to SNC AGs. This method, which is rapidly described below, is simpler, in our opinion, than Filé's one [Fil 87] or Reps' one [Rep 84]. This method is interesting mainly for two reasons :

i) it is based on a natural extension of the classical recursive evaluators for SNC-AGs : in addition to the recursive procedures for computing sets of s-attributes, which walk down the input tree, we construct procedures for computing sets of i-attributes, which use recursion for walking upward in the tree.

ii) as consequence of (i), these procedures can be constructed in a way very close to that of the classical evaluators for SNC AGs.

First, let us explain rapidly what one means by *incremental attribute evaluator* ; for more details see [Rep 84]. In syntax-directed editors, attributes may be used for checking the static semantics of the program being developed. At each moment of the construction of a program, the editor keeps the parse tree t of the (partial) program p developed so far, together with a full and consistent valuation of t. Any transformation of p, either the addition, deletion or modification of a phrase, is viewed by the editor as the replacement of a subtree of t with a new fully and consistently attributed subtree s. In general, some of the attributes of the node u of t at which the replacement takes place, and also of some other nodes, have inconsistent values w.r.t. the resulting tree t'. The task of the incremental evaluator is to compute a new consistent valuation of t' by reevaluating only some of its attributes.

The first step in deriving our incremental method is to describe a method which can compute, in an exhaustive way, all the attributes of t', starting at this particular node u. For this purpose we must find a mean to compute i-attributes of this node. As in the SNC method we will construct procedures computing i-attributes by climbing up in the tree. For this to be possible, the dependency graph $R_{DNC}(X)$ of each non-terminal X must respect two closure conditions, the

cpu_T = CPU time of our transformation, measured in seconds on the Multics system.

The rough results are presented in table 2.

The first remark is that, as announced, our transformation induces a size ratio, K, which is at worst 5% greater than unity (see *Simula* example). The second remark is that the number of constructed tops is the dominant factor of CPU time, independently from the choice of inclusion orderings. Finally the transformation determines that the *Pascal* and *PL/1-C* AGs are l-ordered, but these AGs are not ordered [Kas 80].

	pascal	simula	PL/1-C
NT	114	126	139
prod	214	244	376
atr_{ave}	7.39	7.28	8.90
OAG	**no**	**no**	**no**
DNC	yes	yes	yes
SNC	yes	yes	yes

Table 1 — our AG examples

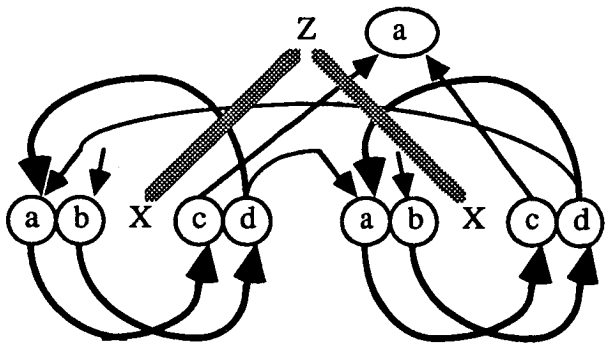
It must be noticed that our transformation will not always detect if the input AG is l-ordered. It will possibly construct an output AG with a size greater than the input AG, even if the latter is l-ordered. However this is not a great loss, as exemplified by the *Simula* result : if this AG was l-ordered (of course we did not test it), the saving on size (1.04 to 1, see table 2) would not be very significant compared to the time required for the l-ordered test .

AG	Pascal			Simula			PL/1-C		
incl. order.	a1	a2	a3	a1	a2	a3	a1	a2	a3
A1 πA_{max}	4	2	1	13	9	4	2	2	1
K	**1.3**	**1.1**	**1**	**3.2**	**2.7**	**1.04**	**1.4**	**1.02**	**1**
B	3.38	3.34	3.24	4.2	4.2	3.7	3.8	3.9	3.7
cpu_T	75	74	71	191	166	133	143	100	123
A2 πA_{max}	3	2	1	12	6	4	2	2	1
K	**1.1**	**1.08**	**1**	**3**	**2.6**	**1.04**	**1.05**	**1.02**	**1**
B	3.18	3.2	3.2	4.08	4.10	3.6	3.48	3.47	3.4
cpu_T	90	86	97	262	241	180	150	147	169

Table 2 — practical results of the SNC to l-ordered transformation

We now make two notes about the problem of finding the best evaluation orders (tops) :

• The average number of blocks B is the double of the average number of visits to each input tree node. If we compare that number B with the average number of attribute per non-terminal (atr_{ave}), we can say that algorithm A1 reduces the visits number to nearly the half ; this is already a good result, because this algorithm was not designed for that purpose but only to be simple. On the contrary, the A2 algorithm, whose aim was to reduce the visits number, shows, when com-



$G(p)\ (top_2 \leftarrow top_1)$

Figure 2 — strong inclusion.

To avoid reexamining all the productions in which the replaced top is involved, we also require that the other tops appearing in these productions be actually unchanged ; this *independence* condition is the second one stated in [Par 88].

Our two inclusion orderings, the *weak inclusion* and the *strong inclusion*, are two particular orderings respecting these conditions. Weak inclusion compares the projections of the tops at the attribute occurrences level ; this is depicted in Figure 1. For strong inclusion, we first check the context condition, and then compare the *contribution* (defined in [Par 88]) of the tops to the rest of the production ; this is depicted in Figure 2. More details can be found in [Par 87, Par 88].

2.3. Practical results.

Let us give some practical results, without much detail, of our inclusion orderings and topological sort. Readers can find more information in [Par 88]. We have implemented this transformation with inclusion orderings and tried it on some realistic examples. In this section we will present some measures we conducted on them.

In tables 1 and 2, we use the following notations :

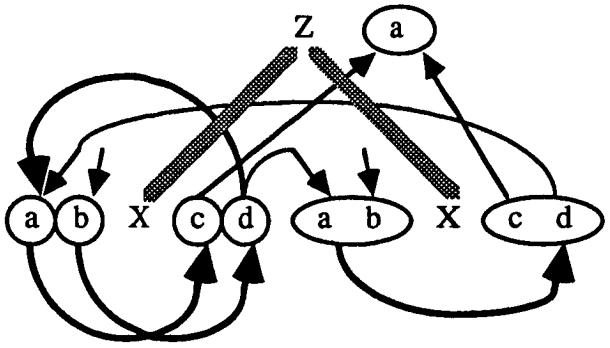
Pascal, Simula, PL/1-C = examples of real AGs with a reasonable size ; more information can be found in table 1.

NT	=	number of non-terminals.
prod	=	number of productions.
atr_{ave}	=	average number of attributes per non-terminal.
SNC, DNC[2], *OAG*	=	answer of the corresponding test.
a1	=	standard transformation.
a2	=	weak inclusion.
a3	=	strong inclusion.
A1, A2	=	two topological sort algorithms.
πA_{max}	=	maximum number of tops per non-terminal .
K	=	average number of tops per non-terminal : **size ratio.**
B	=	average number of attributes blocks.

[2] see [Bar 84, Fil 87] or next section for DNC, and [Kas 80] for OAG.

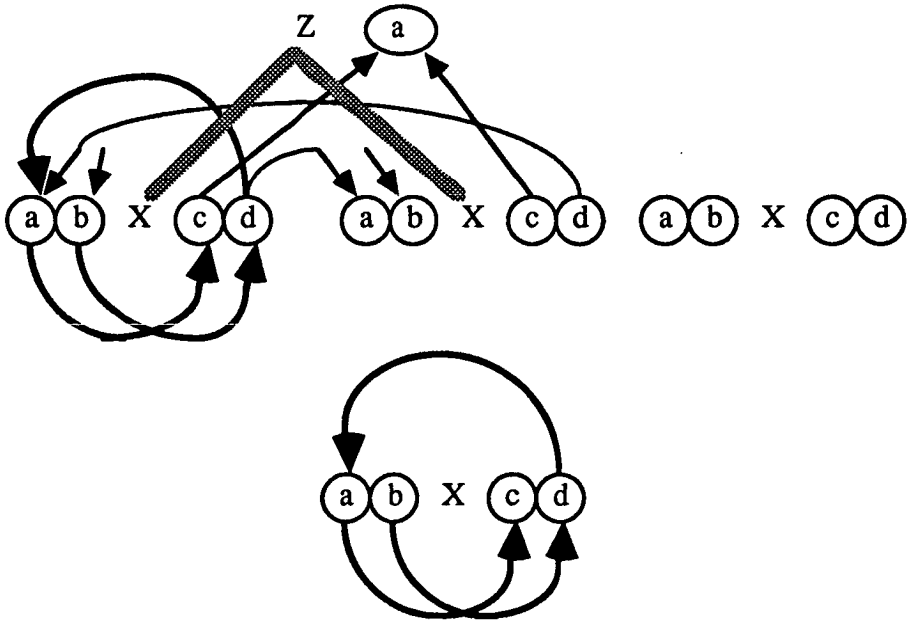
$G(p) = D(p)\ [top_0\ (Z),\ top_1\ (X),\ top_2\ (X)]$

$top_0\ (Z)$



$top_1\ (X)$ $top_2\ (X)$

the projection of $top_2\ (X)$ $\not\subset$ the projection of $top_1\ (X)$



the context condition : $[G(p)\ (top_2 \leftarrow top_1)$ - $top_1\ ]^+_2$ = the empty graph $\subset top_1\ (X)$

the contribution of $top_1\ (X)$ $\subset$ the contribution of $top_2\ (X)$

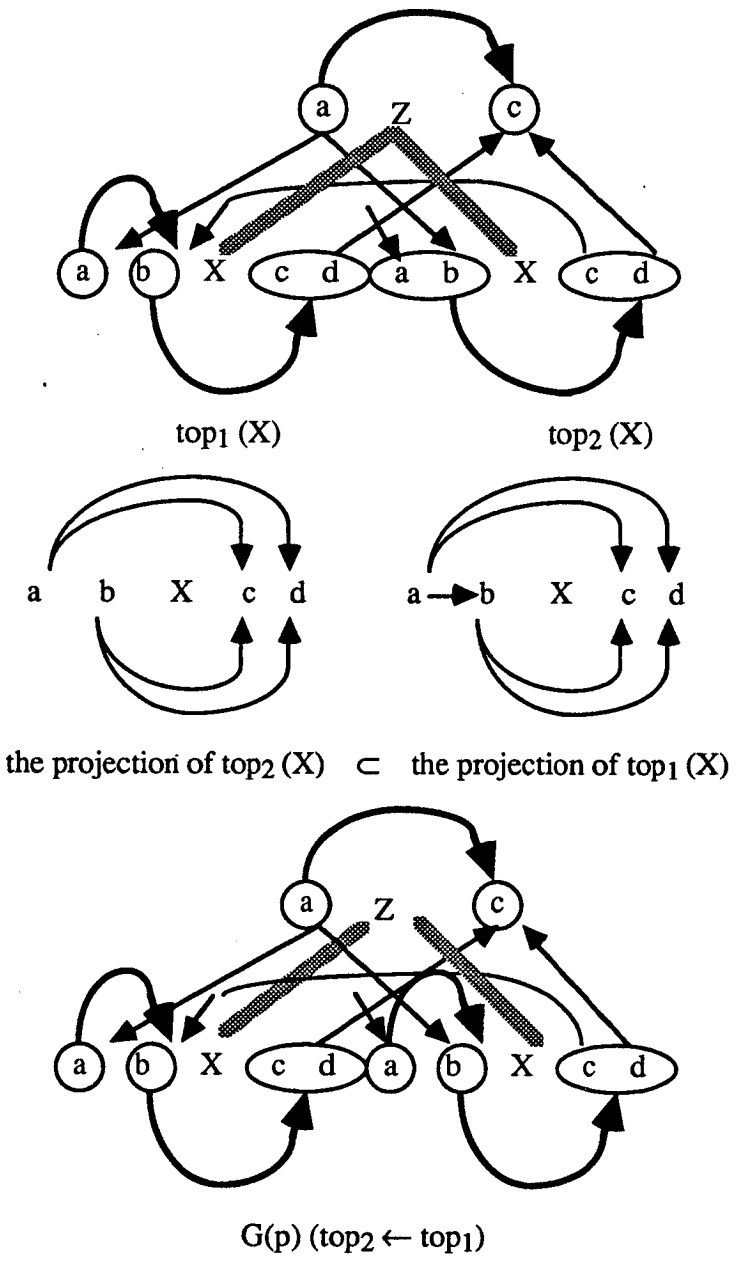


Figure 1 — weak inclusion.

Our inclusion orderings are to this transformation what *covering* [LP 75] is to the non-circularity test, i.e. a means to reduce the number of tops of each non-terminal while preserving the semantics of the construction. They define how a (new or existing) top can be replaced in the construction by another (existing or new) top in the same family. Informally, the replacing top must "have the same effect" in the production as the replaced one, possibly with a slight loss of efficiency (number of visits), which means that the replacing top must respect the *context* in which the replaced top is involved ; this is the first condition stated in [Par 88].

its to each node. And so, in [GB 85], a new characterization of SNC AGs is introduced, where these graphs D(X) are graphs between sets of attributes which respect similar properties. In [Fil 87] these graphs are noted *pops* (partially ordered partitions). It is also proven that finding the best pops — a better pop is one which has the least number of attribute blocks, resulting in the minimum visits number — quickly becomes an NP-complete problem [GB 85, Yel 84].

Secondly, the recursive evaluators described above use a dynamic test for determining whether the attributes they have to compute are already evaluated, because they may be called more than once [Jou 84a]. It is possible to remove this recomputation test only if a total evaluation order is statically known. For that purpose [Rii 83] defines that an AG is SNC iff for each non-terminal X there is a *set* T(X) of graphs which induce a total evaluation order and respect the closure and non-circularity properties. In [Fil 87] these graphs are noted *tops* (totally ordered partitions). Using these tops we can construct completely deterministic evaluators, based on visit sequences, for SNC AGs [Fil 87, Far 84, Rii 83, Yel 84]. In fact this construction is another form of the classical SNC to l-ordered transformation [Der 84]. The other advantage of this characterization is to allow to apply to the SNC class the good features of evaluators associated to l-ordered AGs : memory optimization techniques based on lifetime analysis, implemented in the GAG system [KHZ 82, Kas 84, Kas 87], can be applied only if the attribute evaluation order is statically known (see section 4).

An important problem remains : it has been proven that the size of the l-ordered grammar issued from this transformation could be exponential w.r.t. the size of the input grammar [Fil 83]. More precisely, the number of tops to be constructed on each non-terminal is in the worst case exponential in the number of attributes.

The purpose of our work is to show that this transformation is indeed quite realistic for practical AGs — the size ratio is very close to *one* — because we have introduced two new notions of *inclusion orderings* on tops, noted respectively *weak inclusion* and *strong inclusion*.The practical results for real AG examples are very promising. Our transformation remains exponential in the *worst cases*, but the informal subclass of SNC AGs for which our transformation results in a size ratio close to one constitutes a wide subset, large enough for realistic AG writing. This idea of finding inclusion orderings better than simple equality was suggested by Farrow [Far 84] and, in a somewhat different form, by Deransart [Der 84]. This technique is also a particular case of an approximate solution for a Grammar Flow Analysis problem [Mön 87].

Secondly, for the problem of finding the best top, we have introduced two different topological sort algorithms (almost similar to the one described in [Rii 83]) which allow to construct better tops. However they will not be described in this paper for lack of space ; see [Par 87, Par 88].

2.2. Inclusion orderings.

Let us recall the original transformation of SNC AGs into l-ordered ones (see [Eng 84] for more details). First, the SNC dependency graphs D(X) must be constructed for each non-terminal ; testing strong non-circularity is done during this construction. Then, the set of tops attached to each non-terminal is constructed by an algorithm similar to the classical non-circularity test [Knu 68, LP 75, DJL 84], but working top-down in the grammar instead of bottom-up. More precisely, on each production we choose a top for the LHS non-terminal, construct (using a topological sort algorithm) a new top for each non-terminal in the RHS respecting the required properties, and add those new tops to the corresponding families if necessary ; this is repeated until no new top can be added. Thus the tops play the role of the dependency graphs in the non-circularity test.

l-ordered AGs. Section 3 presents a new method for efficient incremental evaluation of SNC AGs ; incremental evaluation enlarges the application field of AGs to e.g. intelligent editors and programming environments [Rep 84]. Section 4 presents improvements over the (already efficient) space management techniques by Kastens [KHZ 82, Kas 84, Kas 87] which are applicable to visit-sequences evaluators ; they should reduce space consumption — the point which probably most hinders the industrial use of AGs — to a quite acceptable level. Section 5 describes some techniques which greatly improve the construction-time behavior of a number of algorithms participating in the generation of the evaluators, including characterization tests and the above described transformation ; this adds up to the user-friendliness of any AG processing system. Section 6 presents OLGA, a new language designed specially for writing AGs and which features a number of constructs intended to ease the development of large applications. FNC-2 itself is described in section 7, while the next section describes some of its applications.

Because of lack of space, the presentation will be very sketchy ; each section will give references to more detailed publications. For the same reason the reader is assumed to have good knowledge of AGs ; see the seminal paper by Knuth [Knu 68] and the survey by Deransart, Jourdan and Lorho [DJL 88] for more information.

2. TRANSFORMATION OF SNC AGS INTO L-ORDERED ONES

2.1. Introduction.

Most of the methods devised to overcome the attributes evaluation non-determinism (see [Eng 84] and part I of [DJL 88]) are applied at evaluator construction time, but restrict the AG class which they can accept. Others are applied at run-time, thus accepting any AG, but they generally increase the run-time overhead. Two AG classes are particularly interesting in this respect : the class of *l-ordered* AGs (called simple multi-visit in [EF 82] and uniform in [Far 84]), for which the associated evaluation method is deterministic, and the class of *strongly non-circular* AGs [CF 82] (SNC, called absolutely non-circular in [KW 76]). The latter is interesting because quite efficient recursive evaluators can be constructed automatically for any SNC AG [CF 82, Jou 84a] and, moreover, testing whether an AG is SNC takes polynomial time. This class is a very large AG subclass, including all the practical examples, in particular those encountered in compilation. For these reasons SNC AGs have been used in compiler writing systems [Jou 84b] and studied in many theoretical papers [CF 82, Far 84, Fil 87, KW 76, GB 85, Rii 83, Jou 84a]. The l-ordered class, as for it, is the largest class for which the attribute evaluation order is known at evaluator construction time, which implies that we can build efficient evaluators with very good properties ; however the membership test is NP-complete [Fil 83].

There are several characterizations of the SNC class ; some of them describe formally the transformation of SNC AGs into l-ordered ones. Let us recall that the original definition of [KW 76] states that an AG is SNC if, for each non-terminal X, there is a graph D(X) between inherited attributes and synthesized attributes which respects two conditions of non-circularity and closure. With this family of graphs we can construct automatically for each s-attribute of each non-terminal a recursive procedure which computes it (for more details see [Jou 84a]). The evaluator for an SNC AG is thus composed only of a set of recursive procedures for the s-attributes. Generalization of this characterization have been given in [Rii 83, GB 85]. In [Fil 87] we find a good synthesis of all characterizations and the author raises two important questions.

First, it is sometimes possible to compute more than one attribute at the same time (one recursive procedure can compute more than one attribute), in order to minimize the number of vis-

1. INTRODUCTION

Attribute grammars (AGs) have been devised nearly 20 years ago by D.E. Knuth [Knu 68] to describe and implement the semantics of programming languages and, more generally, any syntax-directed computation. The main qualities of this approach are as follows :

- AGs are a *declarative* specification, stating only *what* values the attributes should have in the end and not *how* (i.e. in which order) they are computed ;
- they are a *structured* specification, the structuring base being the productions of the underlying context-free grammar ;
- they feature a strict *locality of reference*, with productions and associated semantic rules acting as independent "black boxes" related only through the attribute set of each non-terminal ;
- they are *executable* specifications : it is possible to thereby generate the attribute evaluator realizing the computation specified by a given AG.

The three former qualities make AGs a very attractive *specification tool*, which allows very impressive productivity gains (decrease of development costs) in areas such as compiler writing[1]. However AGs have not yet encountered the success they deserve as a *programming tool*, in particular in industrial applications, mainly because the efficiency in time and space of the automatically-generated evaluators is still insufficient to compete honorably with their hand-written counterparts, even if they are cheaper to produce. It must be noticed however that constructing efficient evaluators is not a trivial task : firstly, attributes evaluation is a priori non-deterministic, and overcoming this requires computing time ; secondly, the number of attribute instances to compute and store grows rapidly with the size of the input text, and memory management problems arise.

Much work has been done to solve those problems (see [DJL 88]), resulting in, on one hand, the definition of a number of more or less restricted AG subclasses together with associated evaluation methods and, on the other hand, the implementation of a number of AG processing systems realizing more or less efficiently one of these evaluation methods. Some of these systems — in particular HLP78 [RSS 83], LINGUIST [Far 82], MUG2 [GGM 82] and GAG [KHZ 82] — are more than student projects and have honorable performance ; however their reputation is still, apart from rare experiences, confined to the academic world.

This paper presents some recent research work whose goal is, again, to improve the compromise between expressive power, efficiency and versatility of the AG approach. We also present the implementation of their results in FNC-2, a new AG processing system which is still under development and which aims at true production quality.

Section 2 presents an improvement of the well-known transformation of strongly (absolutely) non-circular AGs into equivalent l-ordered ones ; the improvement is such that the size of the resulting AGs, although theoretically exponential w.r.t. that of the original AG, is practically nearly the same ; this combines the expressive power of the SNC class with the efficiency of the deterministic evaluators based on visit-sequences which can be constructed for

---

[1] The first author has written and tested in three months, using the FNC/ERN system, a full ISO-Pascal to P_code compiler [Jou 84b].

THE FNC-2 SYSTEM : ADVANCES IN ATTRIBUTE GRAMMARS TECHNOLOGY‡

LE SYSTEME FNC-2 : DES PROGRES DANS LA TECHNOLOGIE DES GRAMMAIRES ATTRIBUÉES

Martin JOURDAN & Didier PARIGOT

INRIA
Domaine de Voluceau — Rocquencourt
BP 105
78153 LE CHESNAY Cedex
(FRANCE)

E-mail : {jourdan,parigot}@minos.inria.fr

Abstract : This report presents FNC-2, a new attribute grammar (AG) processing system which aims at expressive power, efficiency, ease of use and versatility. These qualities result from a number of recent research findings, also presented in this report, which include practical transformation of strongly non-circular AGs into l-ordered ones, incremental evaluation of SNC AGs, improved space management techniques, construction-time optimizations and a new AG specification language. FNC-2 should therefore reach production quality and demonstrate with industrial applications the usefulness of the AG method.

Keywords : attribute grammars, attribute evaluation, visit sequences, AG transformation, complexity, incremental evaluation, space management, circularity tests, applicative language, modularity.

Résumé : Ce rapport présente FNC-2, un nouveau système de traitement de grammaires attribuées (GAs), dont les buts sont la puissance d'expression, l'efficacité, la facilité d'utilisation et la souplesse d'emploi. Ces qualités découlent d'un certain nombre de résultats de recherche récents, présentés aussi dans ce rapport, dont une transformation pratique des GAs fortement non-circulaires en GAs l-ordonnées, une méthode d'évaluation incrémentale des GAs FNC, des techniques de gestion mémoire améliorées, des optimisations appliquées au cours de la construction et un nouveau langage de spécification de GAs. FNC-2 devrait ainsi atteindre une qualité industrielle et démontrer par des applications "grandeur réelle" l'utilité de la méthode des GAs.

Mots-clés : grammaires attribuées, évaluation d'attributs, séquences de visite, transformation de GAs, évaluation incrémentale, gestion mémoire, tests de circularité, langage applicatif, modularité.

---

‡ Work partially supported by the Commission of the European Communities under ESPRIT Project #956 "COCOS". A preliminary version of this report appeared as one of the deliverables of the COCOS Project.